# A SYSTOLIC ARRAY PARALLELISING COMPILER

*A Thesis Submitted*
*In Partial Fulfilment of the Requirements*
*for the Degree of*
**MASTER OF TECHNOLOGY**

*by*

**C. S. Raghavendra**

*to the*

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
**INDIAN INSTITUTE OF TECHNOLOGY KANPUR**

FEBRUARY, 1992

## CERTIFICATE

It is certified that the work contained in the thesis entitled " A SYSTOLIC ARRAY PARALLELISING COMPILER " , by RAGHAVENDRA, C. S. has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

[illegible]

( Sanjeev Kumar )
Assistant Professor,
Dept of Computer Science and Engineering,
Indian Institute of Technology,
Kanpur - 208 016.

February 1992.

Th
001.642
P1368

CSE-1992-M-RAG-SYS

## ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my thesis supervisor, Dr. Sanjeev Kumar for all his help and consistent encouragement given to me in the course of my thesis.My interaction with Dr. Kumar during this thesis project has created in me a deep interest in this challenging field of compilers. I am grateful to him for this.

My grateful thanks are to Dr. Anil Mahanta, who evinced keen interest in this project and furnished me with the architectural details. I am also indebted to Ajay Sharma who was always there to help me in using the simulator.

My special thanks are to Dr. Subramanya for his invaluable encouragement, Sri. Ananda R., who sat with me throughout and gave valuable suggestions all along and Sri. Shankar, whose extraordinary sense of perfection helped me to bring out this thesis report. Sri. Sudheer and Gautam's proficiency in typing the manuscript was extremely helpful.

I am indebted to Sri. Ananda B. K., Uday, Madhu, Rashi and Mahesh who made a deep impression on me during my stay here.My heartiest thanks to the members of Kannada Sangha, especially to the faculty members who made me feel at home during my stay at IITK. My association with them has been an experience that I would always cherish.

## ABSTRACT

SYSC is a compiler for linear systolic array. SASP is one such linear systolic array under development at IIT, Kanpur. A high-level language called S2 is chosen for programming SASP instead of assembly language used hitherto. SYSC translates programs written in S2 to assembly code of SASP. This thesis discusses the design and implementation of the back end of SYSC.

# CONTENTS

*Chapter* 1. **INTRODUCTION** 1

*Chapter* 2. **ARCHITECTURE OF SASP** 4

2.1 Overview 4

2.2 Intercell Communication 6

2.3 Host 8

2.4 Interface Unit 8

2.5 Cell Unit 10

*Chapter* 3. **MACHINE ABSTRACTION AND PROGRAMMING LANGUAGE S2** 16

3.1 The machine abstraction 16

3.2 The S2 Language 17

*Chapter* 4. **STRUCTURE OF SYSC** 22

4.1 What is SIMPL-S ? 22

4.2 Structure of SYSC 23

4.3 IFU Flow-Graph Extractor 26

4.4 Code Generators 28

*Chapter* 5. **THE CODE GENERATOR** 29

5.1 The Cell Code Generator 29

5.2 The Code Generation Algorithm 36

5.3 Code Generation for IFU 41

*Chapter* 6. **IMPLEMENTATION OF SYSC** 42

6.1 Quadruple Constructor 42

6.2 Representation of Basic Blocks 43

6.3 Data Flow Analyser 43

6.4 Code Generator 44

6.5 Testing SYSC 44

*Chapter* 7. **CONCLUSIONS** 52

7.1 Benefits of SYSC 52

7.2 Quality of the output 53

7.3 Suggestions for future work 56

**REFERENCES**

*Appendix* A. **SYNTAX OF SIMPL-S**

*Appendix* B. **SYSC OUTPUT FOR TEST PROGRAMS**

## LIST OF FIGURES

| | | |
|---|---|---|
| 1-1 | Basic configuration of a Systolic Array | 2 |
| 2-1 | SASP System overview | 5 |
| 2-2 | Data path of Interface Unit | 8 |
| 2-3 | Cell data path | 12 |
| 3-1 | An S2 program for matrix multiplication | 18 |
| 4-1 | SIMPL-S version of the matrix multiplication program of Figure 3-1 | 23 |
| 4-2 | Structure of SYSC | 24 |
| 6-1 | Matrix multiplication program : | |
| (i) | Mapping of input data to the array | 46 |
| (ii) | Computation on the array | 46 |
| 6-2 | Polynomial evaluation program : | |
| (i) | The S2 program | 48 |
| (ii) | Mapping of data to the array | 48 |
| (iii) | Steady state of computation | 48 |
| 6-3 | 1-D convolution program | |
| (i) | The S2 program | 49 |
| (ii) | Mapping of data to the array | 50 |
| (iii) | Computation on the array | 50 |
| 6-4 | S2 program for systolic sorting | 51 |
| 7-1(i) | Typical program written by an assembly language programmer | 54 |
| (ii) | S2 program for the same task | 54 |
| 7-2(i) | Handcoded assembly program for matrix multiplication | 55 |
| (ii) | The microcode output of SYSC for the same | 55 |

# 1. INTRODUCTION

The different generations in computers have been marked by significant leaps in the speed of computation. The earlier generations were characterised by sequential and uniprocessor machines. The ever increasing demand for faster computers could not be matched by comparable advances in electrical technology. Parallel computing was an attractive solution to overcome this barrier.

Several approaches have been taken to build parallel architectures, the choice often guided by the application on hand. Array processing is one such technique, which has gained tremendous recognition in modern signal and image processing applications. A systolic array is a practical realisation of this technique.

The basic configuration of a primitive systolic model is schematically shown in Figure 1-1. This model has a simple synchronous protocol. With each beat of the clock, every cell receives data from its neighbours, computes with the data and outputs the results which are then available as inputs to its neighbours at the next clock.

Most systolic architectures that were built in the early days of systolic computing were highly application specific and were

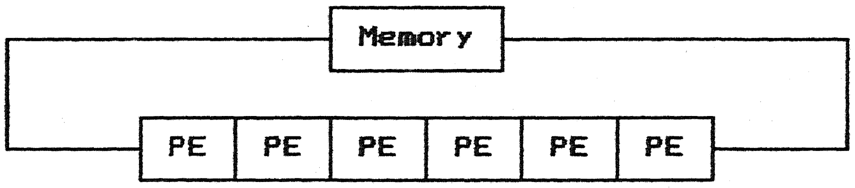


Figure 1-1 Basic configuration of a Systolic Array.

found wanting in generality, in part due to the absence of proper compiler support. In 1985, H T Kung and his group at the Carnegie Mellon University built a high-performance, programmable, general purpose systolic array, called Warp machine [*Anna*]. This was supported by a language called W2 with an optimising compiler . A large variety of systolic algorithms have been implemented on Warp machine using this compiler [*Lam*].

Closer home, at IIT Kanpur, a Systolic Array Signal Processor (SASP) machine is being developed [*Usman*]. A simulator with a meta-assembler is already available [*Wand*] and a large number of algorithms have been implemented on this simulator [*Sharma*]. All these algorithms were handcoded using the micro-instructions of SASP. A mastery of the huge instruction set and a thorough knowledge of even the lower level details of the SASP machine were imperative for the programmer to code these algorithms. Assembly language programming is best avoidable; handcoding in systolic machines is all the more formidable, since excessive importance is given to cell-level details. This sidetracks the more important issue of efficiently mapping the algorithm to the systolic architecture. Hence, we preferred S2, a modified version of W2 as the high-level programming language for SASP. The design and implementation of the back end of a compiler for S2 were the objectives of this thesis. This compiler called SYSC takes a program in S2 and produces the microcode for the simulator.

This dissertation is organised as follows :

In Chapter 2 we present the description of the SASP machine.

Chapter 3 describes the machine abstraction given to the user and the programming language S2.

The SYSC compiler structure is outlined in Chapter 4.

The code generator is explained in Chapter 5.

Chapter 6 describes the implementation issues of SYSC.

Finally, we present the conclusions and list a few suggestions for future work in Chapter 7.

# 2. ARCHITECTURE OF SASP

SASP is a linear systolic array of computational cells. The SASP architecture is similar to Warp architecture. This chapter presents the architecture of SASP in detail. For a compiler writer, a good understanding of the architecture is essential. As far as the user is concerned, the cell-level details are, in a sense, irrelevant and he need only be conversant with a broad overview of SASP. The machine abstraction which hides the cell-level details is dealt with in Chapter 3.

## 2.1 Overview

There are three major components in the system - the SASP array, the interface unit (IFU) and the host [*Usman*] as depicted in Figure 2-1.

The SASP array performs computation-intensive routines such as low-level vision routines or matrix operations. It is a one-dimensional systolic array with identical cells called SASP cells. Data flows through the array on two data paths (X and Y), while addresses (for local cell memories) travel on the Addr path. Control signals from the neighbouring cells are received through Cntrl channel.

The IFU transfers data between the array and the host and regulates the flow of intermediate results through the array. It also generates addresses for local cell memories and systolic control signals which are used in the computation of the cells.

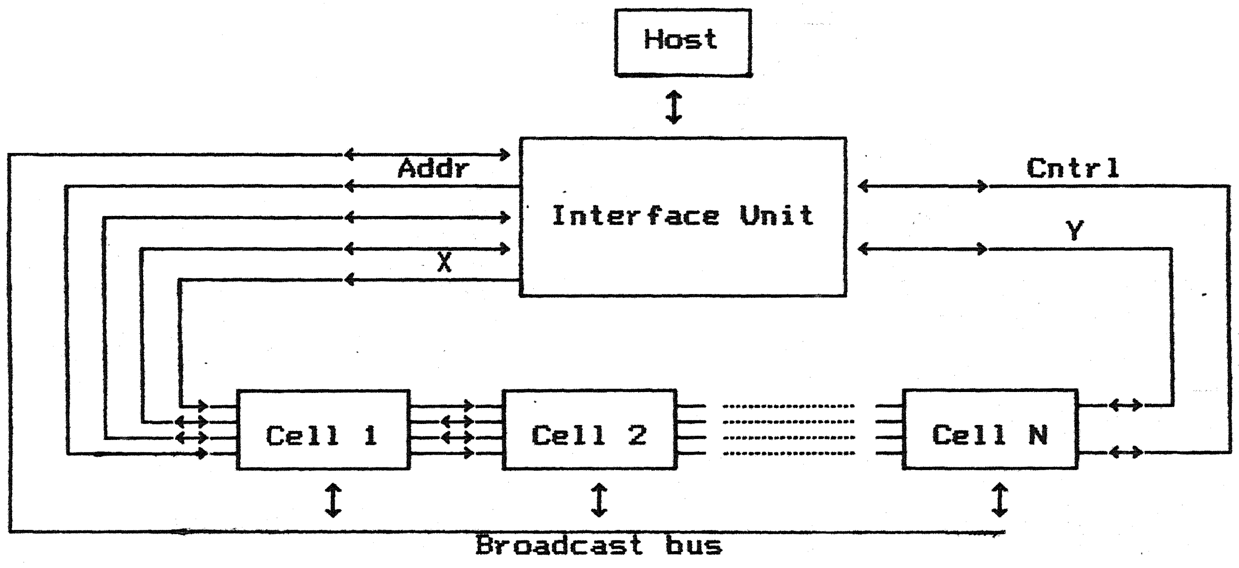


Figure 2-1 SASP System Overview

The host supplies data to IFU and SASP cells and receives the results from the IFU, in addition to executing those parts of the application program that are not mapped onto the SASP array. For example, I/O is handled by the host.

Before the start of execution by the array, the host dumps the data onto the RAMs of the IFU and onto the data RAMs of each cell over the broadcast bus (BC-bus). In the same way, the object code is dumped onto the microcode RAMs of IFU and the cells. After this initial setup, the IFU takes over control of all the blocks. It supervises routing and exchange of data during the execution of the program. After execution is over, it generates an interrupt, asking the host to take necessary action. Then the host may read the results and may process the received results further. This is how a program is typically executed on the SASP system.

## 2.2 Intercell communication

In SASP machine, global communication is only through the BC-bus. However, during execution, only local communication channels are used.

The cells use asynchronous communication primitives for communicating with each other - cells send and receive data to and from their neighbours through dedicated buffers. A receive operation retrieves a data item from the specified channel and stores it in the variable desired. Similarly, a send operation sends the value of the specified variable on the desired channel. A queue is associated with each channel (XQ, YQ and AddrQ) and is placed in the data path of the input cell.

The semantics of asynchronous communication have been supported in SASP through dynamic flow of control. When a cell tries to read from an empty queue, it is blocked (i.e., machine cycles are skipped) until the data item arrives. Similarly, when a cell tries to write into a full queue, it is blocked until a data item is removed from the full queue. Only the cell that tries to read from an empty queue or deposit a data item into a full queue is blocked.

### 2.2.1 Communication channels

The channels are as described below :

**X channel.**

It is a 32-bit wide data path and is unidirectional. It starts from the IFU and ends on the last cell. The data on which computation is to be done is transmitted over this channel and it ripples through the cells without being modified.

**Y channel.**

It is also a 32-bit wide, bidirectional channel and its direction can be statically reconfigured by the microcode. This channel forms a closed loop starting from IFU, running through the processor array and ending at the IFU. Intermediate or final results travel on this channel.

**Addr channel.**

It provides addresses for local data memories in the cells. Data independent addresses are generated in the IFU and transferred along with data on the X-channel. For example, when multiplying two matrices, each cell is responsible for computing a column of the result. All cells access the same memory location which has been loaded with different columns of one of the argument matrices. Therefore, common addresses and loop controlled signals can be generated in the IFU and propagated to all the cells.

**Cntrl channel.**

It contains control signals to read from or write into queues and the status of the queues of the neighbouring cells.

## 2.3 Host

The SASP array is integrated into a general purpose host (PC/XT). The host uses BC_bus to load data and microprograms into each cell.

## 2.4 Interface Unit (IFU)

The block diagram of IFU is shown in Figure 2-2.

The interface to host enables the host to access different parts of the IFU as well as the cells.

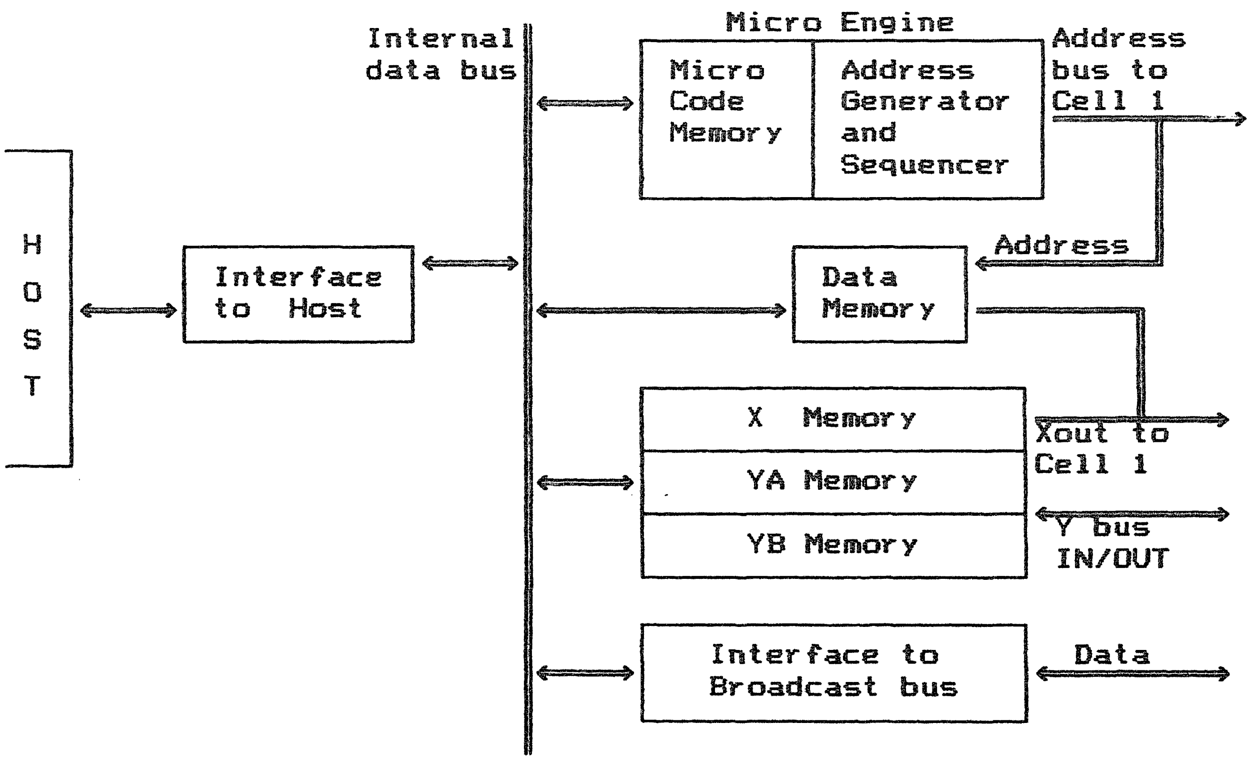


**Figure 2-2 Data path of Interface Unit.**

**Microengine.**

The control unit of IFU is a programmable microengine with a sequencer and an address generator. Control instructions/signals for each device are put together into a wide micro-instruction in central microcode memory and a micro-instruction is executed in every system clock cycle.

During each cycle, the sequencer monitors the conditions and micro-instruction to determine the next microprogram address.

The address generator generates addresses on the address bus which flow systolically from cell to cell. It also has to address the data memory in the IFU. In a single instruction, the device can

- output a 16-bit memory address
- modify this memory address
- detect when the address value has moved to or beyond a preset boundary and conditionally loop back to the top of a circular buffer.

**Memory Structure.**

The interface unit consists of four types of memories.

**X Memory.**

The host can read and write into X memory. The IFU can also write into X memory from the data memory, whenever the previous data loaded is over. A send operation on X channel writes the next data item of X memory into the X queue of cell 1.

**YA Memory.**

The host can read and write into YA memory. This memory can be used for final or initial results. A send operation from the IFU on Y channel writes the next data item of YA memory into Y queue of cell 1 (or cell n depending on Y direction). When cell n (cell 1) executes a send operation on the Y channel, then the value is automatically stored in the next YA memory location.

**YB Memory.**

This is used for storage of intermediate results and can be accessed by the IFU and cell 1 (cell N) at the same time. The send and receive operations are similar to that of YA memory.

The X, YA and YB memories are accessible only sequentially. The fourth memory is data RAM.

## 2.5 Cell Unit

Each SASP cell is a computational unit, having its own sequencer and program memory.

The architecture of each SASP cell is different from that of a conventional processor. Here, all the functional units are spread out to achieve maximum possible parallelism and are connected by a crossbar. The SASP cell has a wide instruction format, in which dedicated fields control the independent functional units.

The data path of a SASP cell is illustrated in Figure 2-3. We explain the salient features of the cell data path below.

**Microengine.**

The structure of the cell microengine is similar to the IFU microengine.

Data independent addresses are generated in the IFU, whereas data dependent addresses are generated in the SASP cells. Thus, the address generator is used as the local address generation unit.

Addresses to local data memory and scratch pad memory are fed from the address cross bar, which has inputs from the address queue and address port of the address generator. The sequencer and address generator data ports get the data from the data cross bar. Thus, the data to these data ports can be from the constant field of the micro-instruction or from any of the inputs to the data cross bar.

In a single cycle, a SASP cell can initiate two floating-point operations, read/write a data item from data

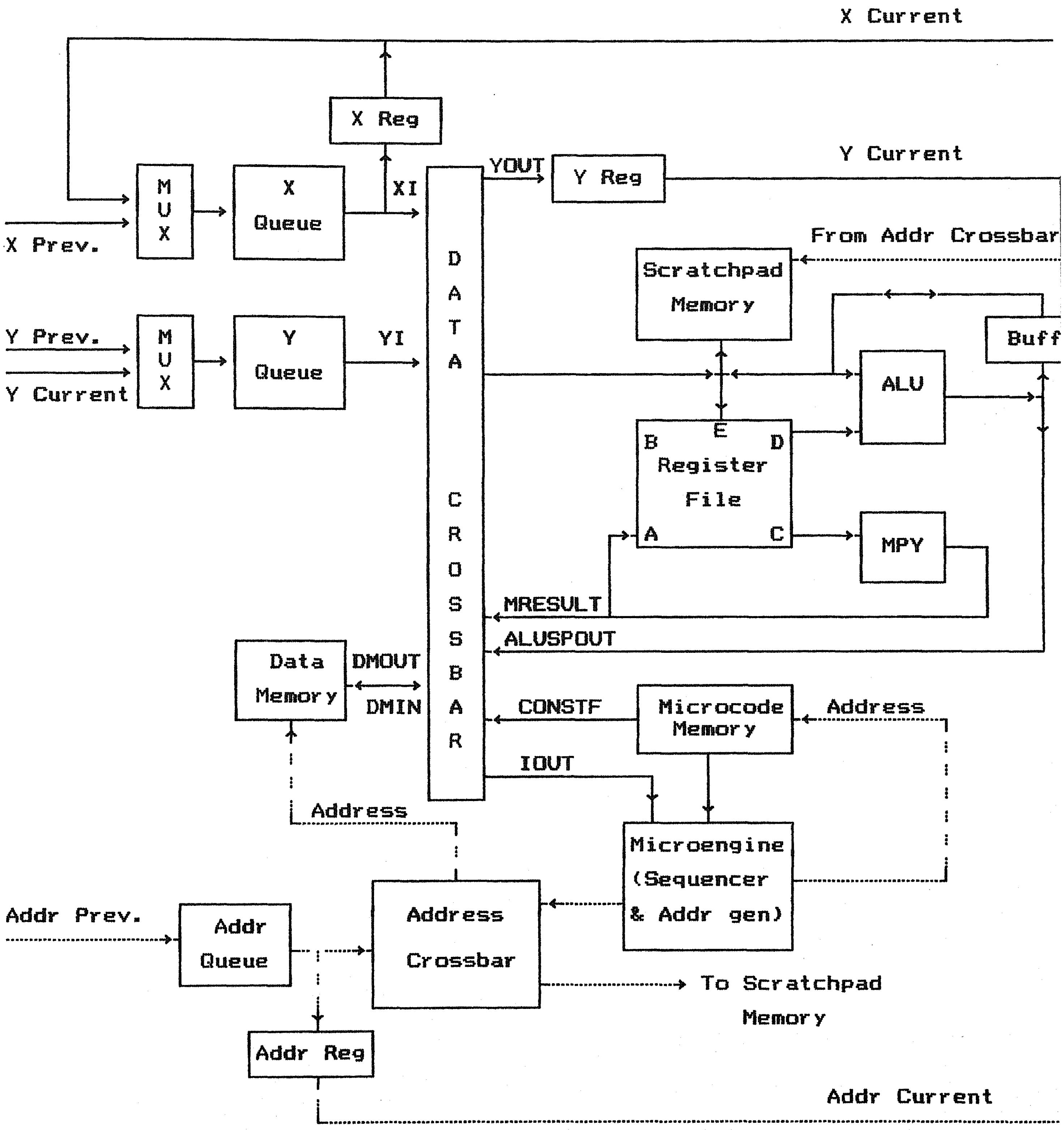


Figure 2-3 Cell Data Path

memory, send and receive two data items from and to its neighbours, read three words and write two words to a register file (or vice versa) and conditionally branch to a program location. The machine has an orthogonal instruction set. This orthogonal instruction set with the long instruction word format exploits maximum parallelism at the instruction level [*Ellis*].

**Inter cell Communication.**

Both the X and Y queues can receive data from the previous cell or from itself. For the Addr queue, no feedback is provided.

Since SASP is a board design, reading of the local queue and writing to the queue of the neighbouring cell in a single cycle is difficult to realise in hardware. So a send operation writes a data item into a queue through a register in two cycles. This explains the presence of X , Y and Addr registers.

**Cross bar.**

Internal data bandwidth is often the bottleneck of a systolic cell. In the cell, the two floating point chips can consume up to four data items and generate two results per cycle. Cross bar connects various data storage blocks supporting the high data processing rate. There are 5 input ports, 4 output ports and one bidirectional port. An output port can output data from any of the inputs, irrespective of the other outputs.

The input ports are

| | | |
|---|---|---|
| XI | --> | from X queue |
| YI | --> | from Y queue |
| constf | --> | from the constant(data) field of the microcode |
| mresult | --> | from the multiplier output |
| alu_spout | --> | from the ALU output |

The output ports are

| | | |
|---|---|---|
| Ain | --> | to ALU input |
| Bport | --> | to B port of the register file |
| iout | --> | to the internal data bus of the microengine (16-bit bus) |
| yout | --> | input to the Y register |

The bidirectional port is to/from the data memory.

**Data storage Units.**

The local data storage units include a data memory, a register file and a scratchpad memory.

**Data Memory.**

The local data memory can be accessed in every clock cycle. It is generally used for storing data transmitted during the execution of a program.

**Register file.**

It contains 128 32-bit wide registers accessible from any of the 5 ports. Two ports are input ports (A and B), two are output ports (C and D) and one (E) is bidirectional.

Register file can be used for storing the intermediate results and to transfer the data/results to/from the various units.

**Scratchpad Memory.**

The scratchpad memory can be used to hold scalars, floating point constants and small arrays. The addition of the scratchpad increases memory bandwidth and improves throughput from those programs operating mainly on local data.

**Computational Units (ALU and Multiplier).**

Both ALU and Multiplier are 2-stage pipelined. Cells in a unidirectional systolic array can be viewed as stages in a pipeline. Thus the processor array supports pipelining at both the array and cell levels. This 2-stage pipelining greatly enhances the system throughput.

The ALU supports addition, subtraction and different kinds of logical operations.

The Multiplier initiates a new multiplication implicitly in every cycle with whatever data is present at its inputs. To execute a multiplication operation, data is given to the multiplier inputs and the result is taken out from the multiplier output after two clock cycles.

This description of the architecture will aid in understanding the actions of the code generator.

# 3. MACHINE ABSTRACTION AND PROGRAMMING LANGUAGE S2

This chapter outlines the machine abstraction for systolic arrays and programming language called S2 which supports the abstraction.

## 3.1 The Machine abstraction

The machine abstraction defines the lowest level details that are exposed to the user of the machine. The main goals of this machine abstraction are generality and efficiency. Hence, the model must allow the user to control the mapping of the computation across the array. We have adopted the machine abstraction proposed by [*Lam*].

The user sees the machine as an array of simple processors communicating through asynchronous communication primitives. We seek to expose the array level parallelism while hiding the parallelism at the cell level. This is not unreasonable since automatic, effective problem decomposition is currently not possible for all computational domains, whereas compiler techniques are superior to hand coding when it comes to generating microcode for highly parallel and pipelined processors.

## 3.2 The S2 language

The abstraction described above is supported by a language called S2. S2 is a modified version of W2 [*Lam*] that was developed for Warp architecture. In this language, each cell in the systolic array is individually programmed in a FORTRAN-like language. Cells communicate with their left and right neighbours via asynchronous communication primitives provided by the language. We wish to emphasize that the user has control only over the array level concurrency and not the system and cell level concurrency.

S2 is a FORTRAN-like language with implicit data typing. The typing rules are as follows-

1. All variables that access the data memory are floating point operands.
2. An operation that involves floating point operands yields a result that is also of type floating point.
3. Any other variable is of type integer.

Thus, loop indices and array subscripts are of type integer. Subscripting is permitted only in a single dimension because of limited addressing capability of the SASP address generator. Further, only addition and subtraction are the permitted operations to calculate the array offsets.

The control flow constructs provided in S2 are **do-enddo**, **if-goto** and **goto**.

Figure 3-1 shows an example of an S2 program that performs matrix multiplication of two matrices of size 3*2 and 2*3.

```
module MatrixMultiply( A,B,C )

DEM   B[2]     /* Each column of B matrix in cell data memory */
XMEM  A[6]     /* matrix A is stored in X memory of IFU        */
YAMEM C[9]     /* Result matrix C will be stored in YA  memory
                * of IFU   */

cell()
{
  do i = 1,3
       row = 0;
      /* each cell computes the dot product of it's column and
       * same  row   of A */
       do j = 1,2
          receive(X,x,A);
          send(X,x);
          row = row + x * B[j];
       enddo;
       receive(Y,temp,C);
       do k = 1,2
          receive(Y,temp,C);
          send(Y,temp,C);
       enddo;
       send(Y,row,C);
  enddo
}
```

Figure 3-1   An S2 program for matrix multiplication

This program will be explained later.

An S2 program is a module. A module has a name, a list of module parameters, and one or more cell programs. The module parameters are like the formal parameters of a function - they define the formal names of the input and output from the array. The user specifies the memory locations of these parameters using reserved words. The SASP machine has four kinds of data memories and the user maps his data to these memories.

| | | |
|---|---|---|
| XMEM | : | X Memory of IFU |
| YAMEM | : | YA memory of IFU |
| YBMEM | : | YB memory of IFU |
| DMEM | : | Data memory of cell |

A cell program describes the actions of a group of one or more cells. Although the same program is shared by a group of cells, it does not necessarily mean that all the cells execute the same instruction at the same time, since a cell cannot start its computation until it has received the input from the preceding cell.

Four types of statements are supported inside a cell program - the assignment, communication, conditional and iterative statements. The assignment, conditional and iterative statements have conventional syntax and semantics. Although iterative constructs (S2 has only do) can be simulated using conditional statements and the infinitely abusable goto, the sequencer of SASP handles them in a different way for reasons to become clear later.

**Communication statements.**

There are two types of communication statements: **receive** and **send**. They are used to specify the interaction among the cells, as well as between the IFU and the end cells of the array.

* **receive (a, b, c)**

  where

  **a:** : channel name (X or Y)
  **b** : a local variable
  **c** : an external (IFU) variable- must be a module parameter.

**receive** removes a data item from **a** and stores it in **b**. The first cell of the array receives data directly from the IFU and the value is explicitly specified by **c**; all other cells receive the data transferred by the corresponding send operation of the communicating cell.

* **send (a, b, c)**

  where

  **a** : channel name (X or Y)
  **b** : a local variable
  **c** : an external (IFU) variable - must be a module parameter

**send** sends the value of **b** on **a**. In addition, the result from the end cell is stored into **c**. **c** is an optional parameter for **send**. For example, common data sent by the IFU and propagated to all the cells need not be stored back on IFU.

The direction of the Y channel is set by a simple assignment statement.

As an aside, it may be mentioned that S2 differs from W2 in the following respects-

* explicit type declaration statements are absent in S2
* memory specification instructions are introduced in S2 to handle the four types of data memories in SASP.

With this description of S2, we next consider the structure of SYSC, which is the topic of the ensuing chapter.

# 4. STRUCTURE OF SYSC.

A compiler consists of two major phases: a front end and a back end. The front end consists of phases to do lexical and syntactic analysis, for creation of a symbol table, semantic analysis and generation of intermediate code. Since these phases depend primarily on the source language and are largely independent of the target machine, emphasis was given to the back end. It is assumed that a front end translates an S2 program into statements in an intermediate language called SIMPL-S. The back end of SYSC takes these statements in SIMPL-S as its input and generates the microcode for the SASP machine.

## 4.1 What is SIMPL-S ?

SIMPL-S is an intermediate language in which the expressions of S2 have been broken down into three-address statements. The control flow of the original S2 program remains unaltered. Hence, SIMPL-S can be considered as an intermediate language of relatively high-level three-address statements. The syntax of SIMPL-S is given in Appendix A.

The SIMPL-S equivalent of the matrix multiplication program explained in the previous chapter (Figure 3-1) is given in Figure 4-1.

```
do i = 1,3
     row = 0
     do j = 1,2
        receive(X,x,A)
        send(X,x)
        t1 = B[j]
        t2 = x * t1
        row = row + t2
     enddo
     receive(Y,temp,C)
     do k = 1,2
        receive(Y,temp,C)
        send(Y,temp,C)
     enddo
     send(Y,row,C)
enddo
```

Figure 4-1 SIMPL-S Version of Matrix Multiplication Program of Figure 3-1

We wish to emphasise that only the *complex* expression,

```
row = row + B[j] * x
```

has been broken down into the three-address statements,

```
t1 = B[j]
t2 = t1 * x
row = row + t2
```

## 4.2 Structure of SYSC

The different phases of the SYSC compiler are shown in Figure 4-2.

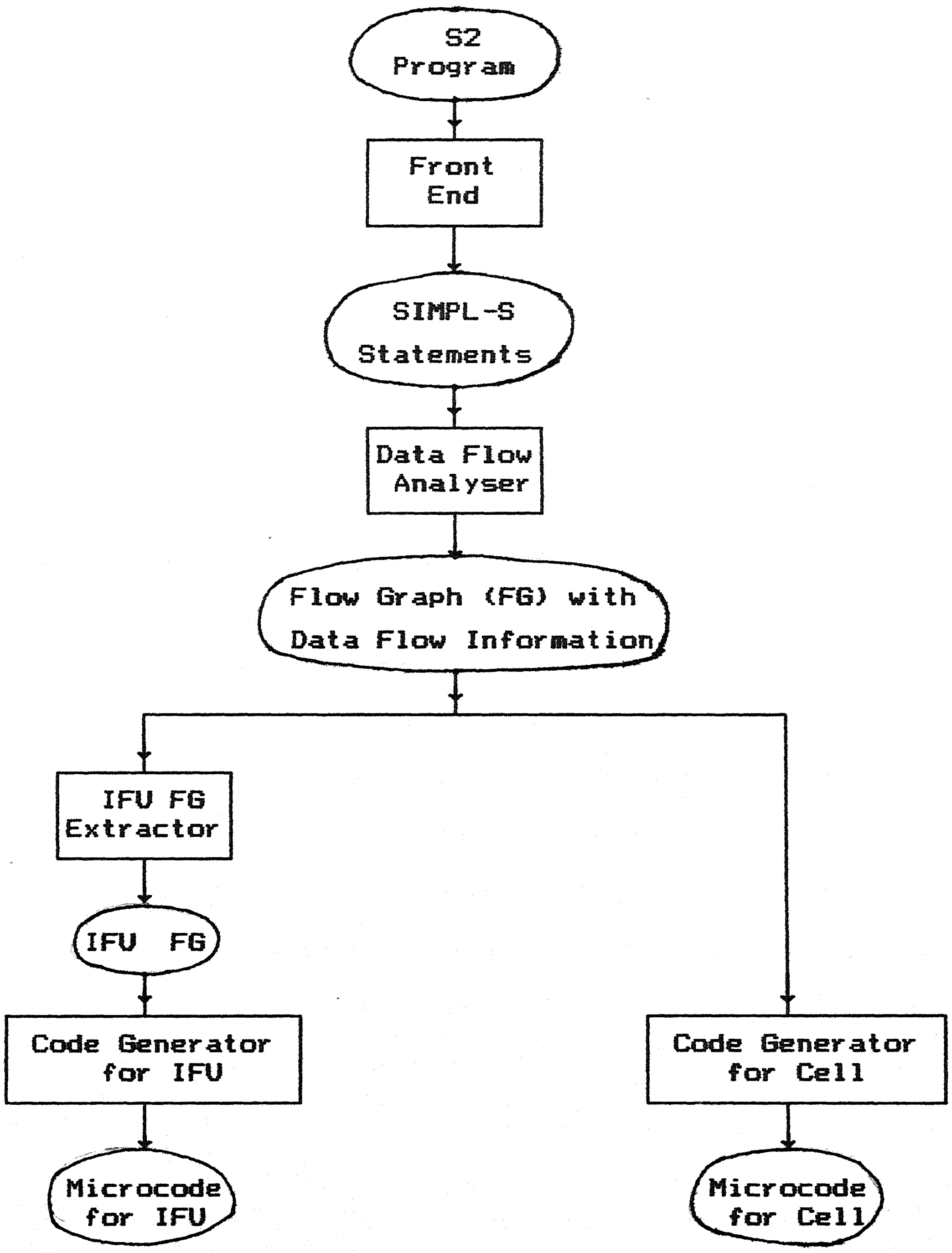


**Figure 4-2 Structure of SYSC**

### 4.2.1 The Front End

Apart from lexical and syntax analysis, this phase stores the three address statements of SIMPL-S as quadruples. Although the front end is not implemented in SYSC, for the purpose of testing the back end, a scanner is written which takes the input in SIMPL-S syntax and outputs it in a form acceptable to the data-flow analyser.

### 4.2.2 The Data Flow Analyser

From the quadruples generated by the front end, a control flow graph of basic blocks is constructed.

For effective code optimization, generation and scheduling, it is imperative that a compiler collect information about the program as a whole and distribute it to each block in the flow graph. Although code optimization is not implemented, the data flow analyzer is designed with this in mind. For an exhaustive treatment of this topic, the interested reader is referred to [*Aho*]. The output of the analyser is a flow graph with data flow information.

We assure the reader that these book-keeping chores are by no means a trivial task. Since the code generator makes extensive use of the information collected by this phase, a significant portion of the thesis work was devoted to a careful design of this phase.

## 4.3 IFU Flow-Graph Extractor

SYSC has to generate separate code for IFU and the cells. The IFU flow graph extractor outputs a separate flow graph for IFU from the flow graph obtained from the data flow analyser. The extractor scans the quadruples in a basic block one at a time and constructs the corresponding IFU quadruple if necessary. The only quadruples of interest are those which could force the IFU to participate in communication with cell 1 (or cell N, depending on the Y direction). All the remaining quadruples are discarded by the extractor. Hence, the actions of the extractor can be classified into the following four cases depending on the input quadruple.

* receive : For every receive encountered, the extractor should generate a corresponding send statement so that at the time of execution, the IFU may send the necessary data to cell 1 (cell N) in the array.

* do : Statement indicating the start of a loop (block).

  The number of receive operations in a cell program should match the number of send operations in the IFU. Hence for a do statement of a block consisting of receive statements, the extractor generates a replica of the do quadruple.

* **enddo** : Statement indicating the end of a loop (block).

  If the corresponding do statement has been replicated in the IFU, so should the **enddo**.

* larray : Statements of the form A[i] = m

  rarray : Statements of the form m = A[i]

  All array references indexed only by expressions of loop indices are considered data independent and are calculated in the IFU. The address calculation in the cell for these array references will be replaced by receive operations from the address queue. For the IFU, the extractor generates a quadruple for **send(AddrQ,i,A)**.

The extractor need not have to generate a **receive** statement corresponding to a **send** statement in the cell N (cell 1) since a **send** operation in the cell N (cell 1) writes automatically into the IFU data memory.

For the semantics of the program to remain valid, the **sends** and **receives** on the two branches of a conditional should follow the same sequence.

## 4.4 Code Generators

The code generators translate the input flow graph into microcode for the corresponding unit, IFU or cell. The detailed description of code generation is the subject matter of the next chapter.

# 5. THE CODE GENERATOR

The final phase of SYSC involves the code generators for IFU and the SASP cell. These code generators translate the flow graph into the corresponding microcode. The microcode is a sequence of very long instruction words, each of which consists of dedicated fields that independently control the functional units of the SASP machine. We first describe the cell code generator in detail. The IFU code generator has a similar but much simpler design.

## 5.1 The Cell Code Generator

We presented the SASP architecture in Chapter 2. Here, we take a closer look at the SASP architecture, since familiarity with the target machine and its instruction set is a prerequisite for developing any code generator.

### 5.1.1 Instructions of SASP

The fields in a SASP cell instruction are -

1. Sequencer
2. Data field (constant)
3. Address generator
4. Flag i/p to sequencer
5. Scratch pad memory

6. X-queue
7. Y-queue
8. Address-queue
9. Data memory
10. Cross bar selection
11. Register file
12. ALU
13. Multiplier

Each unit of the cell is controlled by its instruction field in this long instruction word. For a complete description of the instruction set, the interested reader is referred to the SASP simulator manual [*Wand*].

### 5.1.2 Data movements

Most of the instructions generated by the compiler are used for data transfers in the cell data path. Following are the possible valid data movements in one machine cycle among different units in a SASP cell.

| **Source** | **List of all possible destinations** |
|---|---|
| 1. XQ | X/Y register, ALU inputs, multiplier inputs, data memory, register file input ports |
| 2. YQ | Y register, ALU inputs, multiplier inputs, data memory, register file input ports |
| 3. Data memory | Y register, ALU inputs, multiplier inputs, register file input ports |
| 4. ALUSPOUT | ALU-B input, Y register, multiplier inputs, |

| | |
|---|---|
| | data memory, register file input ports |
| 5. MRESULT | Y register, ALU inputs, multiplier inputs, data memory, register file input ports |
| 6. Register file | Y register, ALU inputs, multiplier inputs, data memory, register file input ports |

Each of these possible data movements is mapped into a set of micro-operations. Throughout the rest of this chapter, we shall refer to such data movements that are possible in one clock cycle as *single-shot transfers*.

## 5.1.3 Address Descriptors

The symbol table entry for each name has an address descriptor field which gives the location where that datum resides. A data item in SASP can reside in any one of the following possible locations.

**1. Data Memory**

Variables that reside in data memory of the cell are specified by the user. So, at the parsing stage itself, we can get the addresses of these names and enter them into their respective address descriptors. These addresses remain the same throughout the program.

**2. Register File**

Unlike conventional machines, the registers in SASP are primarily used for transferring data from the queues (X and Y) and Data memory to the ALU and the multiplier. With 128 registers in the register file, problems like register shortage are rarely encountered, in most applications. For this reason, we have not discussed about register allocation. In fact, for all practical purposes, an infinite register model is assumed.

## 3. Crossbar Input

The SASP code generator does not route a datum through the cell data path unless it is required. The example given below explains the strategy.

Consider the following sequence of statements :

```
  .
  .
m = x * 1
send(Y,m)
  .
  .
```

If m has not been assigned any location, then it can be stored in a register of the register file. Whenever m is used later, we can route it to the required location through the register file. Clearly, this is a simple, but circuitous way of doing things resulting in a lot of redundant data movement and increased code size. For instance, the above example involves two single-shot transfers, one from MRESULT to the register file and another from the register file to YQ. This can be avoided by depositing the multiplier result at the input to the crossbar.

In general, a statement leaves its result at the crossbar input if its address descriptor is empty. To leave a result at the crossbar input, the crossbar input should be free. Otherwise, if it contains a live variable, the variable is stored in a register with its address descriptor appropriately modified.

The list of statements which leave their results at the cross bar input is :

| Statement | Crossbar input |
|---|---|
| 1. receive(X,x) | XI |
| 2. receive(Y,y) | YI |
| 3. a = b+c | ALUSPOUT |
| 4. a = b*c | MRESULT |
| 5. a = m[i] | DMOUT |
| 6. a = k /* k:constant */ | CONSTF |

This necessitates a descriptor for each of the crossbar inputs, pointing to the symbol it contains.

### 5.1.4 Address calculation for data memory elements

If the symbol is a simple variable, then its address in the data memory is directly obtained from its address descriptor.

If the symbol is an element of an array, it can be either data independent or data dependent.

For a data independent address, an instruction to read the address from the address queue has to be generated. Also, the read address has to be written into the address queue of the next cell through another instruction.

A data dependent address will be calculated by the local address generator.

## The local address generator

The local address generator has four sets of registers:

* 16 address(R) registers, to output the calculated address
* 6 offset(O) registers, to store the value of the index
* 4 compare(C) registers, to check the bounds of the array
* 4 initialisation(I) registers, to store the base address of the array

A descriptor table is maintained by the SYSC code generator for each of the above register sets.

Consider the example given below, which illustrates a data dependent address calculation.

```
do i=1,10
      receive(X,x)
      j = i - x
      m = A[j]
        ..
        ..
enddo
```

Since the value of j depends on the value received through the X channel, the address of A[j] is data dependent. The address generation procedure for A[j] involves the following steps :

- Get a new address register, say $R_i$
- Let the base address of A be in $I_j$
  Generate the instruction to move the contents of $I_j$ to $R_i$
- Let the value of j be in $O_k$
  Generate the instruction sequence for $R_i = R_i + O_k$
- Output $R_i$.

### 5.1.5 Register assignment

As mentioned before, the large number of registers in the SASP machine usually preclude the possibility of register shortage. We have further optimized the use of registers by the following strategies :

1. At the end of every basic block, registers containing variables that are dead are freed.
2. Unnecessary storage in the register file is avoided by *lazy (delayed) reading* of the crossbar inputs. This effectively uses the crossbar inputs as temporary storage locations.

### 5.1.6 Sequencer

This has a set of instructions which alter the sequence of the program depending on the flag. It has 4 counter(C) registers to support the do statement.

## 5.2 The Code Generation Algorithm

With this background, we are now ready to tackle the heart of the matter, code generation. The functions and procedures useful for this algorithm are listed below. The main algorithm , *per se*, is then presented.

```
function GET_SRC_ADDR(x)
/* gets the address of x from its address        *
 * descriptor                                    */

function GET_DST_ADDR(x)
/* gets the address of x from its address         *
* descriptor, if it is empty, it selects a        *
* location for x depending on the quadruple and   *
* returns the address of this location            */

procedure SRC_2_DST(src, dst)
/* generates a sequence of single-shot transfer  *
 * instructions to route the data from source to *
 * destination                                   */

procedure  QUAD_CODE(quad)
/* Input  :: A quadruple                          *
 * Output :: A list of instructions              */
```

```
begin
     case (type of quad) of
     send(X,x) :
     begin
         src := GET_SRC_ADDR(x);
                   /* can be in X register   *
                    * or XI                  */
         SRC_2_DST(src, X_register);
         Generate an instruction to write to the  XQ
         of the next cell;
     end;

     send(Y,y) :
     begin
        src := GET_SRC_ADDR(y);
                /* can be a constant,a    *
                 * crossbar input, a data *
                 * memory/register file   *
                 * element                */
        SRC_2_DST(src, Y_register);
        Generate an instruction to write to the  YQ
        of the next cell;
     end;

     receive(X,x) :
     begin
         dst := GET_DST_ADDR(x);
                  /* can be data memory or  *
                   * XI                     */
         SRC_2_DST(XI,  dst);
     end;

     receive(Y,y) :
     begin
         dst := GET_DST_ADDR(y);
                  /* can be data memory or  *
                   * YI                     */
```

```
    SRC_2_DST(YI,  dst);
end;
a[i] = m :
begin
   src := GET_SRC_ADDR(m);
           /* can be a cross bar      *
            * input, a constant or a *
            * register file element   */
   SRC_2_DST(src,DM);
end;

m = a[i] :
begin
  dst   :=  GET_DST_ADDR(m);
            /* can be a register file *
             * element or DMOUT        */
  SRC_2_DST(DATA_MEMORY, dst);
end;

a = b ∘ c :
        /* ∘ is an ALU operation          */

begin
   src  :=  GET_SRC_ADDR(b);
             /* can be a constant,a    *
              * crossbar input, a data  *
              * memory/register file   *
              * element             */
   SRC_2_DST(src,ALU_AINPUT);
   src := GET_SRC_ADDR(c);
   SRC_2_DST(src,ALU_BINPUT);
   generate the code for the operation ∘ ;
   dst = GET_DST_ADDR(a);
          /* can be a register file *
           * element or data memory *
           * or ALUSPOUT             */
   SRC_2_DST(ALUSPOUT,dst);
end;
a = b * c :
begin
   src  :=  GET_SRC_ADDR(b);
               /* can be a constant,a     *
                * crossbar input, a data *
                * memory/register file    *
                * element                 */
   SRC_2_DST(src, MUL_AINPUT);
   src := GET_SRC_ADDR(c);
   SRC_2_DST(src, MUL_BINPUT);
   dst = GET_DST_ADDR(a);
            /* can be a register file *
             * element or data memory *
             * or MRESULT             */
   SRC_2_DST(MRESULT,dst);
end;

do i=1,n :
begin
   Get a counter in sequencer, say C_i;
   Generate the instruction to load C_i with
   the loop count;
   If i is used later for address calculation,
   store it in an offset register of Address
   generator;
end;

enddo :
begin
   Generate the instruction to decrement the
   counter associated with the present loop;
end;
```

```
            if cond then block_num :
            begin
                Generate the instructions for ALU to
                perform the conditional operation;
                Generate the instruction for the sequencer
                to select cond and perform the associated
                action;
            end;

            goto block_num :
                 Generate the instruction for sequencer to
                 jump to block_num unconditionally;
        end.
```

THE MAIN ALGORITHM :

```
        while ∃ a block do
        begin
                for ∀ quads in the block do
                    QUAD_CODE(quad);
                free the address descriptors of dead
                    variables;
                store the remaining variables at the
                    crossbar inputs in the register file;
                move the loop independent instructions
                    outside the block;
        end.
```

The cell code generator generates the microcode for each cell in the SASP array. The microcode of all the cells except the end cell has an additional instruction at the end to wait for the completion of computation on the end cell.

## 5.3 Code Generation For IFU

The code generation for IFU is simple since it does not involve any cross bar and functional units (ALU and multiplier). It just has additional data memory in the form of XMEM,YAMEM and YBMEM.

The only extra quadruple in IFU is a quadruple of the form **send(AddrQ, i, A)**. This quadruple occurs in a do loop. Let us say the index of the do loop is i. The IFU has to send the data independent address of A[i] to cell 1 through the AddrQ. Thus, the actions of the code generator for this quadruple are -

* get a new address register, $R_i$
* Generate the instruction to store the base address of array A in $R_i$ and move it outside the present block
* Generate the instruction to send the value in $R_i$ to the address queue of cell 1.

Since the updated address has to be sent in each pass of the do loop , the IFU code generator generates instructions to increment the value of all data independent addresses (here A[i]) when it encounters a enddo quadruple.

Except for this address generation issue, the IFU code generator follows the same outlines as that of cell code generator.

# 6. IMPLEMENTATION OF SYSC

We enumerate below certain implementation related details of the SYSC compiler.

## 6.1 Quadruple Constructor

In the absence of a full-fledged front end, for purposes of testing the back end, a simple scanner for SIMPL-S statements has been written. A symbol table entry for an identifier has the following structure :

* **name** : name of the identifier
* **type** : type of the identifier
* **val** : value of the identifier, if any
* **defset** : set of all definitions of the variable
* **useset** : set of all uses of the variable
* **addr** : address descriptor

The information about every SIMPL-S statement is stored in a separate node whose contents include :

* **Quad** : the associated quadruple
* **Dflow** : data flow information
* **Instr** : pointer to the set of instructions generated for the quad.

Quadruples are constructed at the time of parsing. A quadruple is a record structure with four fields, which we denote by op, arg1, arg2 and result. The op field contains an internal code for the operator. The other fields point to the symbol table entries of the corresponding data elements.

## 6.2 Representation of Basic Blocks

Each basic block contains a doubly linked list of quadruples. This representation is ideal since data flow analysis involves both backward and forward motions over the flow graph. In addition, the transformations on basic blocks (not implemented in SYSC) involve the movement of quadruples, their deletion and insertion. A basic block entry contains pointers to the leader and trailer quadruples and a list of its predecessor and successor basic blocks.

## 6.3 Data Flow Analyser

The data flow analysis implementation adheres to the approach taken in [*Hecht*]. The iterative algorithms for data flow analysis make extensive use of set manipulation. Thus, an efficient representation for a set would be a sequence of contiguous bytes, where the first byte stores the size of the set and the remaining bytes actually represent the set. The data flow information associated with a node are :

* reaching definition set
* live variable set
* definition number and du-chain, for the result field of quadruple
* use-number1, ud-chain1 and next-use1, for the arg1 field of quadruple
* use-number2, ud-chain2 and next-use2, for the arg2 field of quadruple

## 6.4 Code Generator

The implementation of the code generator involved excessive book-keeping tasks. Since much care was taken during the design of the code generation algorithm, the implementation issues were covered by it. An exhaustive list of data transfers in the cell data path were mapped to the corresponding instruction sequence of SASP. In effect, the code generation algorithm itself is independent of the instruction set of SASP.

## 6.5 Testing SYSC

The SYSC compiler was developed on a SUN3 machine. The code was written in C and ran into nearly 6000 lines.

To test SYSC, we selected a few algorithms and wrote programs for them in SIMPL-S. These programs were compiled using SYSC. As mentioned earlier, SASP is under development but the simulator is available. The output of SYSC was the microprogram for the meta-assembler of the simulator. The microprograms were assembled using the meta-assembler.

Using simulator commands, we loaded the data into IFU memories and the data memory of the cell. Then the object code output of the meta-assembler was run on the simulator.

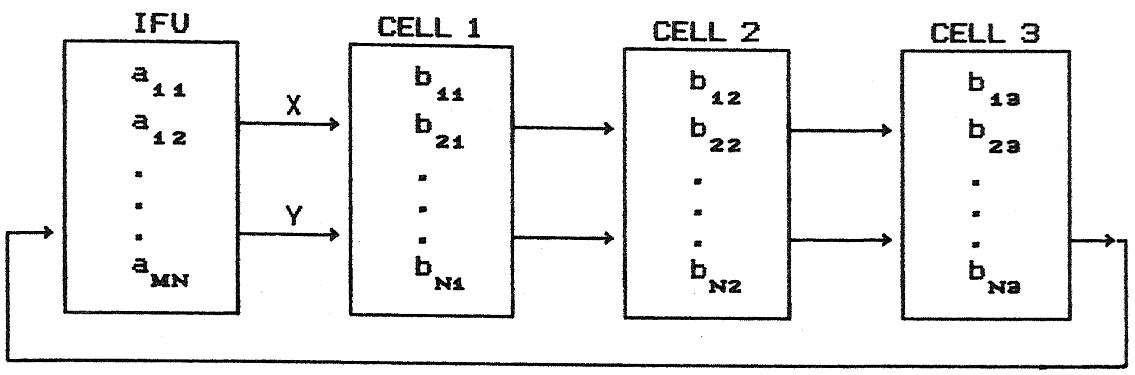
We describe below the S2 programs for which the above test was done.

### 6.5.1 Matrix multiplication program

The S2 program for this is given in Figure 3-1.

Let A and B be two matrices of size M*N and N*K respectively. Matrix A is stored in the X memory of the IFU row by row. Matrix B is stored in the data memory of the cells, one column per cell (we have a total of K cells). Matrix C is stored in the YB memory of the IFU, with zeros as the initial values. The mapping of the array elements is shown in Figure 6-1(i).

The IFU sends the X memory data on X channel sequentially. A cell N computes a column of the result matrix using this X data on the X channel and sends the result to the cell N+1 on Y channel.


IFU
CELL 1
CELL 2
CELL 3
$a_{11}$ $a_{12}$ . . . $a_{MN}$
X
Y
$b_{11}$ $b_{21}$ . . . $b_{N1}$
$b_{12}$ $b_{22}$ . . . $b_{N2}$
$b_{13}$ $b_{23}$ . . . $b_{N3}$

(i) Mapping of input data to the array

M = 3    N = 2    k = 3

iteration = 1 ;   channel = Y

| Cell 1 | Cell 2 | Cell 3 |
|---|---|---|
| $\rightarrow$ temp = $C_{11}$ = 0 | $\rightarrow$ temp = 0 | $\rightarrow$ temp = 0 |
| $\rightarrow$ temp = $C_{12}$ = 0 $\rightarrow$ | $\rightarrow$ temp = 0 $\rightarrow$ | $\rightarrow$ temp = $C_{11}$ $\rightarrow$ |
| $\rightarrow$ temp = $C_{13}$ = 0 $\rightarrow$ | $\rightarrow$ temp = 0 $\rightarrow$ | $\rightarrow$ temp = $C_{12}$ $\rightarrow$ |
| row = $C_{11} = A_{11} * B_{11}$ | row = $C_{12} = A_{11} * B_{12}$ | row = $C_{13} = A_{11} * B_{13}$ |
| $+ A_{12} * B_{21}$ $\rightarrow$ | $+ A_{12} * B_{22}$ $\rightarrow$ | $+ A_{12} * B_{23}$ $\rightarrow$ |

( ii ) Computation on the array

**Figure 6-1** Matrix multiplication program

At the end of computation of an element of a column, cell N receives K elements from cell N-1 on Y channel. Out of this, it sends K-1 elements unchanged to cell N+1 and sends the element computed by it as the Kth element. The same action is performed by all the elements in the array and the IFU gets one row of result matrix at the end of sends by cell K. Figure 6.1(ii) illustrates this. The arrows on the left and right of the statement indicate the receive and send operations respectively.

The receives are ahead of sends by a count of 1. The microcode is analysed in Chapter 7.

### 6.5.2 Polynomial evaluation

Suppose we want to evaluate the polynomial,

$$P(x) = C_m x^m + C_{m-1} x^{m-1} + \ldots + C_1 x + C_0$$

By Horner's rule, the polynomial can be reformulated from a sum of powers into an alternating sequence of multiplications and additions :

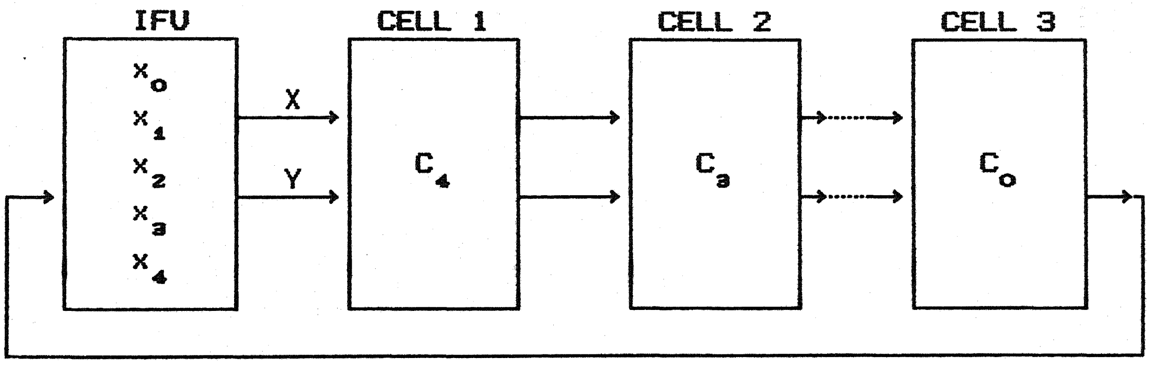
$$P(x) = ((C_m x + C_{m-1})x + \ldots + C_1)x + C_0$$

The S2 program for solving polynomials of degree 4 is given in Figure 6-2(i). The values of x are stored in the X memory of the IFU. The coefficients are stored in the cells, one per each cell (see Figure 6-2(ii)). The mode of computation is straightforward. The steady state of the computation is depicted in Figure 6-2(iii). The SYSC output for this program is given in Appendix B.

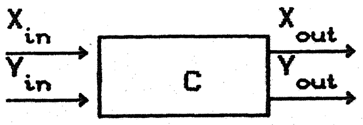
```
DMEM  C       /* coefficient stored in the cell */
YAMEM A[5]    /* values of the polynomial */
XMEM  B[5]    /* different x values */

cell()
{
  /* Compute the polynomials */
  do i=1,5
       receive(X,x,B);
       receive(Y,y,A);
       partial = y * x + C;
       send(X,x);
       send(Y,partial,A);
  enddo;
}
```

( i ) The S2 program



(ii) Mapping of data to the array



$X_{out} = X_{in}$

$Y_{out} = Y_{in} * X_{in} + C$

(iii) Steady state of Computation

**Figure 6-2** Polynomial Evaluation Program

### 6.5.3 Convolution of two sequences

Let us consider the convolution of two sequences x(n) and h(n),

$$Y(i) = \quad x(k) * h(i-k) \qquad 1 <= i < 2n - 1$$

The S2 program for finding the convolution of two sequences, x(3) and y(3) is given in Figure 6-3(i). The input sequence x(n) is kept in X memory and the weights h(n) are kept in the cell array as shown in Figure 6-3(ii). The initial results are kept in YA memory. In this case, these are all zeros.

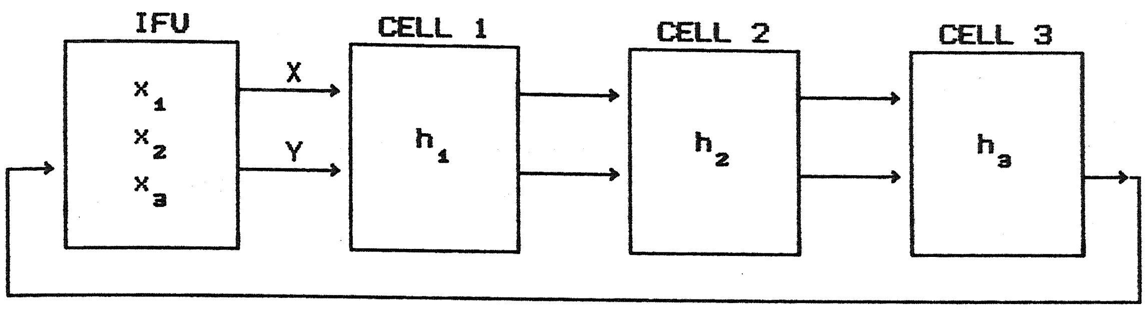
The computation of the result is illustrated in Figure 6-3(iii).

```
XMEM  x[3]  /* the sequence x[n] */
YAMEM y[5]  /* the convolution output */
DMEM  h     /* an element of the second sequence, h[n] */

cell()
{
  receive(Y,y1,y);
  do i=1,3
       receive(X,xdata,x);
       send(X,x);
       receive(Y,ydata,y);
       sum = h * xdata + ydata;
       send(Y,sum,y);
  enddo;
  receive(Y,ydata,y);
  send(Y,ydata,y);
  send(Y,y1,y);
}
```

(i) The S2 program

Figure 6-3 1 - D Convolution Program



(ii) Mapping of data to the array

| Cell 1 | Cell 2 | Cell 3 |
|---|---|---|
| → Y,y1 = $y_1$= 0 | → Y,$y_1$= $h_1 * x_1$ | → Y,$y_1$= $h_1 * x_2 + h_2 * x_1$ |
| → X,xdata=$x_1$→ | → X,xdata=$x_1$→ | → X,xdata=$x_1$ |
| → Y,ydata=$y_2$= 0 | → Y,ydata=$h_1 * x_2$ | → Y,ydata = $h_1 * x_3$ + $h_2 * x_2$ |
| Y,sum = $h_1 * x_1$→ | Y,sum = $h_1 * x_2$ + $h_2 * x_1$→ | Y,sum = $h_1 * x_3$ + $h_2 * x_2 + h_3 * x_1$→ |
| → X,xdata=$x_2$→ | → X,xdata=$x_2$→ | → X,xdata=$x_2$ |
| → Y,ydata=$y_3$= 0 | → Y,ydata=$h_1 * x_3$ | → Y,ydata = $h_2 * x_3$ |
| Y,sum = $h_1 * x_2$→ | Y,sum = $h_1 * x_3$ + $h_2 * x_2$→ | Y,sum = $h_2 * x_3$ + $h_3 * x_2$→ |
| → X,xdata=$x_3$→ | → X,xdata=$x_3$→ | → X,xdata=$x_3$ |
| → Y,ydata=$y_4$= 0 | → Y,ydata=0 | → Y,ydata = 0 |
| Y,sum = $h_1 * x_3$→ | Y,sum = $h_2 * x_3$→ | Y,sum = $h_3 * x_3$→ |
| → Y,ydata=$y_5$= 0 → | → Y,ydata=0 → | → Y,ydata = $h_1 * x_1$→ |
| Y,$y_1$= 0 → | Y,$y_1$= $h_1 * x_1$→ | Y,$y_1$= $h_1 * x_2$ + $h_2 * x_1$→ |

(iii) Computation on the array

Figure 6-3 1 - D Convolution Program (Contd.)

### 6.5.4 Sorting a sequence

```
DMEM  C       /* element retained in the cell */
YAMEM A[5]  /* the unsorted sequence */
cell()
{
        C = 0;

        /* retain the largest element in the sequence */
        do i=1,5
             receive(Y,y,A);
             if C > y goto 20;
             send(Y,C,A);
             C = y;
             goto 30;
         20: send(Y,y,A);

    30: enddo;
}
```

Figure 6-4. S2 program for systolic sorting

We want to sort a sequence y(n) whose elements are greater than 0 into non-decreasing order. Before computation begins, a zero is kept in the data memory of each cell. y(n) is stored initially in the YA memory. As it passes through the array, each cell retains the largest element in the sequence received and sends the remaining values to the next cell. At the end of the computation, the ith element in the sorted sequence is found in cell i. The S2 program to achieve this is given in Figure 6-4 and the SYSC output is given in Appendix B.

# 7.CONCLUSIONS

A systolic array compiler called SYSC has been designed and its back end implemented. Systolic algorithms which are mapped onto SASP can be written in a high-level language called S2. The assumed front end translates these S2 programs into equivalent statements in an intermediate language called SIMPL-S from which the back end takes its input. We conclude the discussion on SYSC with a fair assessment of the work done and suggestions for future work.

## 7.1 Benefits of SYSC

The current style of programming on SASP is coding in assembly language. For this purpose, the programmer has to acquire a firm grasp over the machine details and master the instruction set. A lot of effort is thus spent on these details rather than concentrating on the more important issue of mapping the algorithm onto the SASP array. With the availability of SYSC, the programmer is relieved of the onerous task of understanding the machine details. With only the SASP array architecture visible to him, programming in S2 is much more intuitive.

For example, the assembly language written by a typical programmer without SYSC's support is as shown in Figure 7-1(i). Figure 7-1(ii) shows an S2 program for the same task.

It is seen from this figure that in the S2 program, there is no separate program for the IFU. SYSC generates the code for IFU after extracting the relevant parts from the data flow graph constructed by it.

The listing of some S2 programs is given in Appendix B.

## 7.2 Quality of the output

The code generated by SYSC is efficient to a certain degree. As explained in Chapter 5, SYSC's code uses minimal number of data transfers, i.e. one-shot transfers on the shortest possible path. Thus, the code resembles that which would have been written by an experienced microcode programmer who is ignorant of the principles of compaction.

We give a comparison of a *typical* hand coded microprogram *with the output of SYSC for a corresponding S2 program.* The matrix multiplication microprogram given in Figure 7-2(i) is from [*Wand*].

```
 wrcntr(C1) & 2 8000h+M-2 & ken & xbiout_constf & rst
LOOP1: wrcntr(C0) & 2 8000h+N-3 & ken & xbiout_constf
 yrtr(R0) & dsel & 2 0 & ken & xbiout_constf
 2 0 & ken & xbain_constf & reg_ewr & e_addr(#20h) &
 rdxq & xbbport_xi & b_addr(#0) & reg_bwr & c_addr(#0) &
  reg_crd & mul_aen
 wrxq & yinc(C0)(R0) & rddm & xbbport_dmout & b_addr(#1) &
 reg_bwr & c_addr(#1) & reg_crd & mul_ben & e_addr(#20h) &
 reg_erd & alu_ben & d_addr(#20h) & reg_drd & alu_aen
LOOP2:  dccntr(C0) & rdxq & xbbport_xi & b_addr(#0) & reg_bwr &
 c_addr(#0) & reg_crd & mul_aen & sadd
 wrxq & yinc(C0)(R0) & rddm & b_addr(#1) & xbbport_dmout &
 reg_bwr & c_addr(#1) & reg_crd & mul_ben
 jda(sign) LOOP2 & ken & xbiout_constf & moen & aoen &
 aout_to_in &
 aluout_bufen & alu_ben & d_addr(#20h) & reg_drd &
 alu_aen & a_addr(#20h) & reg_awr
 cont & sadd
 cont
 moen & aoen & aout_to_in & aluout_bufen & alu_ben &
 d_addr(#20h) & reg_drd & alu_aen & a_addr(#20h) & reg_awr
 sadd
 cont
 dccntr(C1) & aoen & xbyout_aluspout
 jda(sign) LOOP1 & ken & xbiout_constf & wryq
 wrcntr(C1) & 2 K*M-cell_no*M-1 & ken & xbiout_constf
LOOP3: dccntr(C1)
 jda(sign) end1 & ken & xbiout_constf
 rdyq & xbyout_yi
 jda(unconditional) LOOP3 & ken & xbiout_constf & wryq
end1: cont

 ;Program for Matrix multiplication. (IFU)
 M equ 3
 N equ 2

 wrcntr(C0) & 2  8000h+M*N/2-2 & ken & clrx & rdx
loop: dccntr(C0) & rdx & wrxq
 jda(sign) loop & ken & wrxq & rdx
 jpcnf
```

(i) Handcoded assembly program for matrix multiplication

```
block1: wrcntr(C0) & 2 8000h+3-2  & ken & xbiout_constf
block2: wrcntr(C1) & 2 8000h+2-2  & ken & xbiout_constf
 2 0  & ken & xbbport_constf & reg_bwr & b_addr(#0h)
block3: reg_crd & c_addr(#1h) & mul_aen & xbbport_dmout & reg_bwr &
 b_addr(#1h) & rddm & xbyout_dmout & axbdm_addrqi & rdaddrsq
 wraddrsq
 rdxq & reg_crd & c_addr(#2h) & mul_ben & xbbport_xi &
 reg_bwr & b_addr(#2h)
 cont
 cont
 wrxq
 reg_drd & d_addr(#0h) & alu_aen
 moen & reg_erd & e_addr(#3h) & alu_ben & reg_awr & a_addr(#3h)
 sadd
 cont
 aoen & xbbport_aluspout & reg_bwr & b_addr(#0h)
 dccntr(C1)
 jda(sign) & 2 block3  & ken & xbiout_constf
block4: wrcntr(C1) & 2 8000h+2-2  & ken & xbiout_constf
 rdyq
block5: rdyq & xbyout_yi
 wryq
 dccntr(C1)
 jda(sign) & 2 block5  & ken & xbiout_constf
block6: reg_erd & e_addr(#0h) & xbyout_aluspout & aluout_bufen &
 ain_to_out
 wryq
 dccntr(C0)
 jda(sign) & 2 block2  & ken & xbiout_constf

/* Sysc generated code for the IFU */

block1: wrcntr(C0) & 2 8000h+3-2  & ken
 2 0  & ken & yrtr(r0) & dsel
block2: wrcntr(C1) & 2 8000h+2-2  & ken
block3: wraddrsq & yrtr(r0)
 rdx
 wrxq
 yinc(c0)(r0)
 dccntr(C1)
 jda(sign) & 2 block3  & ken
block4: rdya
 wryq
 wrcntr(C1) & 2 8000h+2-2  & ken
block5: rdya
 wryq
 yinc(c0)(r0)
 dccntr(C1)
 jda(sign) & 2 block5  & ken
block6: dccntr(C0)
 jda(sign) & 2 block2  & ken
```

(ii) The microcode output of SYSC for the same

**Figure 7-2**

```
HIS PROGRAM IS FOR MATRIX MULTIPLICATION ON SYST. ARRAY.
T SHOULD BE LOADED ON EACH CELL. NO. OF CELLS EQUAL TO K
A_matrix M*N
B_matrix N*K
 equ 2
 equ 3
 equ 3
ell_no equ 1


rcntr(C1) & 2 8000h+M-2 & ken & xbiout_constf & rst
OP1: wrcntr(C0) & 2 8000h+N-3 & ken & xbiout_constf
rtr(R0) & dsel & 2 0 & ken & xbiout_constf
 0 & ken & xbain_constf & reg_ewr & e_addr(#20h) &
dxq & xbbport_xi & b_addr(#0) & reg_bwr & c_addr(#0) &
reg_crd & mul_aen
rxq & yinc(C0)(R0) & rddm & xbbport_dmout & b_addr(#1) &
eg_bwr & c_addr(#1) & reg_crd & mul_ben & e_addr(#20h) &
eg_erd & alu_ben & d_addr(#20h) & reg_drd & alu_aen
OP2:  dccntr(C0) & rdxq & xbbport_xi & b_addr(#0) & reg_bwr &
_addr(#0) & reg_crd & mul_aen & sadd
rxq & yinc(C0)(R0) & rddm & b_addr(#1) & xbbport_dmout &
eg_bwr & c_addr(#1) & reg_crd & mul_ben
a(sign) LOOP2 & ken & xbiout_constf & moen & aoen &
ut_to_in &
uout_bufen & alu_ben & d_addr(#20h) & reg_drd &
u_aen & a_addr(#20h) & reg_awr
nt & sadd
nt
en & aoen & aout_to_in & aluout_bufen & alu_ben &
_addr(#20h) & reg_drd & alu_aen & a_addr(#20h) & reg_awr
dd
nt
cntr(C1) & aoen & xbyout_aluspout
a(sign) LOOP1 & ken & xbiout_constf & wryq
cntr(C1) & 2 K*M-cell_no*M-1 & ken & xbiout_constf
P3: dccntr(C1)
a(sign) end1 & ken & xbiout_constf
yq & xbyout_yi
a(unconditional) LOOP3 & ken & xbiout_constf & wryq
1: cont

rogram for Matrix multiplication. (IFU)
 equ 3
 equ 2

cntr(C0) & 2  8000h+M*N/2-2 & ken & clrx & rdx
p: dccntr(C0) & rdx & wrxq
a(sign) loop & ken & wrxq & rdx
cnf
```

i)  Typical program written by an assembly language programmer

```
module MatrixMultiply( A,B,C )
DEM  B[2]       /* Each column of B matrix in cell data memory */
XMEM A[6]       /* matrix A is stored in X memory of IFU       */
YAMEM C[9       /* Result matrix C will be stored in YA  memory
                 * of IFU    */
cell()
{
  do i = 1,3
       row = 0;
      /* each cell computes the dot product of it's column and
       * same  row    of A */
       do j = 1,2
          receive(X,x,A);
          send(X,x);
          row = row + x * B[j];
       enddo;
       receive(Y,temp,C);
       do k = 1,2
          receive(Y,temp,C);
          send(Y,temp,C);
       enddo;
       send(Y,row,C);
  enddo
}
```

(ii)  S2 program for the same task

**Figure 7-1**

We observe the following differences.

* The hand coded program does not utilise the full power of the machine. In the above hand coded program, the address computation for data independent addresses are computed in each cell, which reduces the speed of computation. This is the case with most of the hand coded microprograms on SASP [*Sharma*]. In the SYSC output, address generation for data independent addresses are computed in the IFU and propagated through the address channel to all the cells, reducing the computational burden on the cell.

* The code size produced by SYSC is larger and takes more time. This is so, because of the absence of a scheduler. With a scheduler, we can get as good a code as a hand coded one. We do not envisage any conceptual difficulty in the implementation of a scheduler for SYSC - because of a time crunch, we could not do it in this thesis.

## 7.3 Suggestions for future work

To be sure, this compiler is not complete in all respects. For SYSC to become completely operational, a front end and a scheduler are mandatory. More parallelism than what SYSC extracts can be obtained by overlapping the operations from different basic blocks. The WARP compiler, for instance, employed the techniques of software pipelining and hierarchical reduction for its scheduler([*Lam*]). The interested reader is also referred to [*Fisher*]'s dissertation, for a comprehensive evaluation of scheduling techniques.

W2 and S2 are high-level programming languages, which hide the cell architecture from the user but expose the array architecture. In these languages, the user programs each cell individually and manages inter cell communication explicitly. Mapping scientific programs to programs in the above languages is a nontrivial task and an area of ongoing research. In fact, a compiler is being developed ([*Tseng*]) for a high-level language AL, in which the user views the entire systolic array as a sequential machine and the compiler generates W2 programs with inter cell communication. Although this compiler extracts the parallelism from only a few programs, a similar task for SASP would be an interesting exercise.

## REFERENCES

[Aho] Aho, A. V., R. Sethi and J. D. Ullman, "Compilers: Principles, Techniques and Tools", Addison-Wesley, 1986.

[Anna] Annaratone, M. and E. Arnould, "The Warp Computer Architecture, Performance", IEEE Transactions on Computers, Vol. C-36, No.12, Dec 1987.

[Ellis] Ellis, J. R., "Bulldog: A Compiler For VLIW Architectures", MIT Press, 1986.

[Fisher] Fisher, J. A., "The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Appliication of Processor Scheduling with Resources", PhD thesis, New York University, Oct., 1979.

[Hecht] Hecht, M. S., "Flow Analysis of Computer Programs", North-Holland, 1977.

[Lam] Lam, M. S., "A Systolic Array Optimising Compiler", Kluwer Academic Publishers, 1989.

[Sharma] Sharma, A., "The Design and Implementation of Algorithms for SASP", M. Tech. Thesis, 1991, I.I.T. Kanpur.

[Tseng] Tseng, Ping-Sheng, "A Systolic Array Parallelising Compiler", Kluwer Academic, 1988.

[Usman] Usman, Mohd., "Design of SASP- A Systolic Array Signal Processor", M.Tech. thesis, I.I.T. Kanpur, 1989.

[Wand] Wandhekar, S. A., "A Simulator For A Systolic Array Signal Processor(SASP) on ADSP-14XX and 32XX chip set", M.Tech. thesis, I.I.T. Kanpur, 1990.

# APPENDIX A. SYNTAX OF SIMPL-S.

The syntax of SIMPL-S is given below in BNF notation.

```
dir_stmt        ::= ydir = LEFT
                  | ydir = RIGHT


statement       ::= labelled_stmt
                  | assign_stmt
                  | cond_stmt
                  | commn_stmt
                  | iter_stmt


labelled_stmt   ::= label statement


assign_stmt     ::= var = - arg
                  | var = arg
                  | var [ arg ] = arg
                  | var = var [ arg ]
                  | var = arg op arg


cond_stmt       ::= var = arg rel_op arg
                  | var = ! arg
                  | var = logical_const
                  | goto integer
                  | if arg rel_op arg goto integer
                  | if arg goto integer


commn_stmt      ::= send ( X , var )
                  | send ( Y , float_arg, var )
                  | receive ( X , var , var )
                  | receive ( Y , var , var )
```

```
iter_stmt       ::= do var = 1, integer stmt_list enddo

rel_op          ::= & | | | < | <= | > | >= | ==

op              ::= + | - | * | /

arg             ::= var | integer | float

float_arg       ::= var | float
```

# APPENDIX B. SYSC OUTPUT FOR TEST PROGRAMS

## 1. Matrix multiplication program

/* Results for the matrix multiplication program given in * *
Figure 7-2 (ii) */

/* Command file for SIMS */

The input matrices are

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \qquad B = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}$$

/* Running the Simulator */

```
Script started on Mon Jan 13 15:31:56 1992
$ sims
_Arch_file: syst.arh

SIM> cmdfile
_File: mat.cmd

SIM> run
Program terminated.

SIM> yb1 -f
rdcntr1=0
wrcntr1=9
_Address:
         0000h : 6.000000e+00
         0001h : 6.000000e+00
         0002h : 6.000000e+00
```

```
        0003h : 1.400000e+01
        0004h : 1.400000e+01
        0005h : 1.400000e+01
        0006h : 2.200000e+01
        0007h : 2.200000e+01
        0008h : 2.200000e+01


SIM> exit
_Verify: Y
$ exit
script done on Mon Jan 13 15:32:48 1992
```

**2. 1-D Convolution problem**

```
/* S2 program for the convolution problem */

   XMEM  x[3]  /* the sequence x[n] */
   YAMEM y[5]  /* the convolution output */
   DMEM  h     /* an element of the second sequence, h[n] */

   cell()
   {
     receive(Y,y1,y);
     do i=1,3
          receive(X,xdata,x);
          send(X,x);
          receive(Y,ydata,y);
          sum = h * xdata + ydata;
          send(Y,sum,y);
     enddo;
     receive(Y,ydata,y);
     send(Y,ydata,y);
     send(Y,y1,y);
   }
```

```
/* SYSC output for the convolution program given above */

/* Code for the cell */

block1: wrcntr(CO) & 2 8000h+3-2  & ken & xbiout_constf
     2 0  & ken & xbiout_constf & yrtr(r0) & dsel
     rdyq & xbbport_yi & reg_bwr & b_addr(#0h)
block2: rdxq & xbbport_xi & reg_bwr & b_addr(#1h)
     wrxq
     reg_crd & c_addr(#2h) & mul_aen & xbbport_dmout & reg_bwr &
     b_addr(#2h) & rddm & yrtr(r0)
     reg_crd & c_addr(#1h) & mul_ben
     cont
     cont
     moen & reg_drd & d_addr(#3h) & alu_aen & reg_awr & a_addr(#3h)
     rdyq & reg_erd & e_addr(#4h) & alu_ben & xbbport_yi & reg_bwr &
     b_addr(#4h)
     sadd
     cont
     aoen & xbyout_aluspout
     wryq
     dccntr(CO)
     jda(sign) & 2 block2  & ken & xbiout_constf
block3: rdyq & xbyout_yi
     wryq
     reg_erd & e_addr(#0h) & xbyout_aluspout & aluout_bufen &
     ain_to_out
     wryq


/* Code for the IFU */

block1: rdya
     wryq
     wrcntr(CO) & 2 8000h+3-2  & ken
block2: rdx
     wrxq
     rdya
     wryq
     dccntr(CO)
     jda(sign) & 2 block2  & ken
block3: rdya
     wryq


/* Running on the simulator */

data x[3] = 1 2 3

     h[3] = 2 2 2


Script started on Fri Jan 24 12:18:13 1992
$ sims
_Arch_file: syst.arh
```

```
/* SYSC output for the convolution program given above */

/* Code for the cell */

block1: wrcntr(CO) & 2 8000h+3-2  & ken & xbiout_constf
      2 0  & ken & xbiout_constf & yrtr(r0) & dsel
      rdyq & xbbport_yi & reg_bwr & b_addr(#0h)
block2: rdxq & xbbport_xi & reg_bwr & b_addr(#1h)
      wrxq
      reg_crd & c_addr(#2h) & mul_aen & xbbport_dmout & reg_bwr &
      b_addr(#2h) & rddm & yrtr(r0)
      reg_crd & c_addr(#1h) & mul_ben
      cont
      cont
      moen & reg_drd & d_addr(#3h) & alu_aen & reg_awr & a_addr(#3h)
      rdyq & reg_erd & e_addr(#4h) & alu_ben & xbbport_yi & reg_bwr &
      b_addr(#4h)
      sadd
      cont
      aoen & xbyout_aluspout
      wryq
      dccntr(CO)
      jda(sign) & 2 block2  & ken & xbiout_constf
block3: rdyq & xbyout_yi
      wryq
      reg_erd & e_addr(#0h) & xbyout_aluspout & aluout_bufen &
      ain_to_out
      wryq


/* Code for the IFU */

block1: rdya
      wryq
      wrcntr(CO) & 2 8000h+3-2  & ken
block2: rdx
      wrxq
      rdya
      wryq
      dccntr(CO)
      jda(sign) & 2 block2  & ken
block3: rdya
      wryq


/* Running on the simulator */

data x[3] = 1 2 3

     h[3] = 2 2 2


Script started on Fri Jan 24 12:18:13 1992

$ sims

_Arch_file: syst.arh
```

```
                goto 30;
            20: send(Y,y,A);

       30: enddo;
   }


/* SYSC output for the sorting program given above */
/* Code for the Cell */

block1: 2 0  & ken & xbiout_constf & yrtr(r0) & dsel
     2 0  & ken & xbdmin_constf & wrdm & axbdm_agi & yrtr(r0)
     wrcntr(C0) & 2 8000h+5-2  & ken & xbiout_constf
block2: reg_drd & d_addr(#0h) & alu_aen & xbbport_dmout & reg_bwr &
     b_addr(#0h) & axbdm_agi & yrtr(r0) & rddm
     rdyq & reg_erd & e_addr(#1h) & alu_ben & xbbport_yi &
     reg_bwr & b_addr(#1h)
     scomp
     cont
     jda(flag) & greaterthan & 2 block4  & ken & xbiout_constf
block3: rddm & xbyout_dmout & yrtr(r0)
     wryq
     reg_erd & e_addr(#1h) & xbdmin_aluspout & aluout_bufen &
     ain_to_out & wrdm & axbdm_agi & yrtr(r0)
     jda(unconditional) & 2 block5  & ken & xbiout_constf
block4: reg_erd & e_addr(#1h) & xbyout_aluspout & aluout_bufen &
     ain_to_out
     wryq
block5: dccntr(C0)
     jda(sign) & 2 block2  & ken & xbiout_constf

/* Code for the IFU */

block1: wrcntr(C0) & 2 8000h+5-2  & ken
block2: rdya
     wryq
block3: dccntr(C0)
     jda(sign) & 2 block2  & ken


/* Running the code on simulator */



Unsorted sequence = A = {3, 6, 15, 12 ,9}
Script started on Fri Jan 24 12:51:27 1992
$ sims
_Arch_file: syst.arh
```

```
SIM> cmdfile
_File: sort.cmd

SIM> run
Program terminated.

SIM> element
Current Command cell: 5
_Number: 1

SIM> dm -f
_Address:
         0000h : 15.000000e+00

SIM> element
Current Command cell: 1
_Number: 2

SIM> dm -f
_Address:
         0000h : 12.000000e+00

SIM> element
Current Command cell: 2
_Number: 3

SIM> dm -f
_Address:
         0000h : 9.000000e+00

SIM> element
Current Command cell: 3
_Number: 4

SIM> dm -f
_Address:
         0000h : 6.000000e+00
```

```
        cont
        cont
        moen & reg_drd & d_addr(#2h) & alu_aen & reg_awr & a_addr(#2h)
        reg_erd & e_addr(#3h) & alu_ben & xbbport_dmout & reg_bwr &
        b_addr(#3h) & axbdm_agi & yrtr(r0) & rddm
        sadd
        cont
        wrxq
        wryq
        dccntr(C0)
        jda(sign) & 2 block2  & ken & xbiout_constf


/* Code for the IFU */
block1: wrcntr(C0) & 2 8000h+5-2  & ken
block2: rdx
        wrxq
        rdya
        wryq
        dccntr(C0)
        jda(sign) & ken & block2
/* Running the code on simulator */


Input x[] = 0,1,2,3,4

Polynomial = x^4 + 2x^3 + 3x^2 + 4x^1 + 5

Script started on Fri Jan 24 11:37:00 1992
$ sims
_Arch_file: syst.arh

SIM> cmdfile
_File: pol.cmd

SIM> run
Program terminated.
SIM> yb1 -f
rdcntr1=0
wrcntr1=5
_Address:
            0000h : 5.000000e+00
            0001h : 1.500000e+01
            0002h : 5.700000e+01
            0003h : 1.790000e+02
            0004h : 4.530000e+02
```

```
SIM> exit
_Verify: y
$ exit

script done on Fri Jan 24 11:37:28 1992
```